Preprint
UCRL-JC-131396

A facility for creating Python extensions in C++

P.F. Dubois

This paper was prepared for submittal to the
Seventh International Python Conference
Houston, Texas, November 9-13, 1998

July 14, 1998

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

A facility for creating Python extensions in C++

Paul F. Dubois

Seventh International Python Conference
November 9-13, 1998
Houston, Texas

Abstract

Python extensions are usually created by writing the glue that connects Python to the desired new functionality in the C language. While simple extensions do not require much effort, to do the job correctly with full error checking is tedious and prone to errors in reference counting and to memory leaks, especially when errors occur. The resulting program is difficult to read and maintain. By designing suitable C++ classes to wrap the Python C API, we are able to produce extensions that are correct and which clean up after themselves correctly when errors occur. This facility also integrates the C++ and Python exception facilities.

This paper briefly describes our package for this purpose, named CXX. The emphasis is on our design choices and the way these contribute to the construction of accurate Python extensions. We also briefly relate the way CXX's facilities for sequence classes allow use of C++'s Standard Template Library (STL) algorithms on C++ sequences.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

The software described herein has been released for unlimited redistribution; please see the detailed legal information that accompanies the software for further information.

1.0 Correctness is the problem we wanted to solve

Anyone who has extended Python using C knows that it is not hard to get something simple working fast. There is even a small Tk-based GUI that will write the framework of a C extension which you can just fill in. However, once you attempt to write an extension of any significant size or complexity, you quickly find it is difficult to maintain correctness, and that the main size and complexity of your program quickly becomes dominated by the extensive coding required for detecting errors and maintaining correct reference counts.

We felt that a suitable set of C++ classes could be used to alleviate these difficulties. In particular, C++'s facilities for defining behaviors during object creation, copying, and assignment allow us to get reference counting right, and C++'s behavior when a C++ exception is thrown is ideal for solving the problem of cleaning up temporary objects when an error is detected. C++ ensures that each local object in the function where the exception is thrown will be properly destroyed, and so on up the call chain until either the job terminates or the exception is "caught". In our case, we will catch the exceptions thrown in Python extensions and convert them into Python exceptions, thus giving the Python user a chance to regain control.

A secondary goal was to bring the power of the STL to Python container classes. By defining a suitable facility, we are able to directly apply STL algorithms (such as *sort*) to Python sequences, and to use iterators over Python sequences so that Python extensions written using C++ look like C++ programming usually looks.

2.0 Each Python type is wrapped by a class

The CXX class hierarchy is rooted in class Object, in namespace Py. An instance of Object holds a pointer to a Python object and owns a reference to that object. When this instance is destroyed, the destructor decrements the Python object's reference count. An Object can be created that holds a pointer to any kind of Python object. The Object class contains methods that correspond to each of the general Python operations. Binary operators such as plus are defined so that we have access to those behaviors for Objects. Stream output is defined to use the object's string representation str().

Sometimes we want to work with the properties of Python objects that are specific to their actual type. We define descendants of Object for each of the Python types; these more specific classes test the Python object to which they are going to point to make sure it is a proper member of the desired type. When this test fails, an exception is thrown. Thus, creating an instance of one of these more specific types not only gives us access to the API appropriate to such an object, it also acts as a runtime type-check.

Here is a simple example. The following program fragment is part of the test routine for the Dict class. Dict, List, String, and Int are CXX classes which correspond to Python's dictionary, list, string, and integer objects.

```
Dict a, b;
List v;
String s("two");
```

```
a ["one"] = Int(1);
a [s] = Int(2);
a["three"] = Int(3);

v = a.values ();
sort (v.begin (), v.end ());

b = a;
b.clear();
```

Here we see that a Python dictionary object *a* is created, and three key/value pairs are added to it. Then the list of values in extracted into a Python list object, and the list object is sorted using the STL algorithm *sort*. Finally, a second reference *b* to the dictionary is created, and the dictionary-specific method *clear* is called to empty it.

Note that at any point if an error had occurred in one of the underlying calls to the Python C API, the result would have been the propagation of an exception. In the act of leaving the routine that contains the above fragment, the objects we had created would be correctly destroyed. If we were to write the above program in C, it would be much, much longer and to correctly identify at each possible point of error, which Python objects need to be destroyed before returning, would be a terrible problem.

2.1 Special facilities are provided for sequences

Note that in the previous example that we were able to use the STL algorithm sort on an instance of class List. This is possible because each of the classes in CXX that corresponds to a sequence type in Python has been declared to inherit from a special templated class SeqBase<T>. Here the template parameter T is intended to be some descendent of Object which defines the most specific type expected for the element of the sequence. Of course, for the standard Python sequences, this type is Object itself: that is, the most we know about an element of a Python list is that it is an Object. Since this is such a common case, the name Sequence is available as a short-hand for SeqBase<Object>.

SeqBase<T> defines a number of facilities that assist the C++ programmer in dealing with sequences. These are:

- Access to sequence elements using the usual array notation works as expected: s[i] is the i'th element of the sequence.
- Contained classes iterator and const_iterator are defined for read/write and read-only access to traversals. For example, to print out each item of a list using stream I/O, we might do:

  ```
  List mylist;
  mylist = ... some list ...
  for(List::iterator j = mylist.begin(); j != mylist.end(); ++j) {
      cout << *j;
  }
  ```

If you are not familiar with STL iterators, this doubtless looks very strange to you. Think of an iterator as a kind of pointer that when incremented knows how to advance itself to the next item in a container. Traditional pointers can only do this with contiguous data structures. Iterators are thus a generalization of C pointers.

- Proper inheritance and internal definitions are given to make SeqBase<T> a proper STL random-access container, thus allowing use of the full range of STL algorithms.

3.0 Creating objects

Each of the CXX classes has a variety of constructors available, appropriate to the particular type. For example, class Float instances can be constructed from C doubles:

```
double d;
Float x(d); // make a Float whose value is d
```

Each of the classes has a constructor which accepts an existing PyObject*. The class instance increments the reference count on the object, thus giving itself an owned reference to the object. When the instance is destroyed, its destructor decrements the reference count.

When the Python object is created by a call to the Python C API the result is often an owned reference already. In that case we want the resulting CXX class instance to take over this ownership. (Since we wrap a great deal of the Python C API in CXX classes, this is more of an issue within CXX's implementation than it is for the end user.) To this end CXX defines a helper class FromAPI whose net effect is to decrement the reference count on a PyObject*. For example, PyDouble_FromDouble is a routine in the Python C API that returns an owned reference to a Python float object. Thus, were we to do:

```
double d;
Float x (PyDouble_FromDouble(d));     //Incorrect
```

the reference count would be incorrect. (Of course, we would not normally do this since the constructor Float x (d) would do the job much more simply.) Instead, we should write:

```
Float x(FromAPI(PyDouble_FromDouble(d))); // Correct
```

3.1 No illegal objects permitted

It is part of the philosophy of CXX that no illegal objects are ever permitted to exist. Each and every CXX constructor must end with its pointer pointing to a legitimate Python object acceptable to that class. This was a decision we came to after trying it both ways. The rule has the virtue of eliminating any concern about whether an object represents a legitimate object; it is the equivalent of eliminating the possibility of a dangling pointer. No consumer of CXX ever need test an object to see if it is a legitimate representative of its class, because it is or its constructor would have thrown an exception.

The downside of this decision is a burden on those extending CXX to add new types. Essentially, one must be able to construct a legal member of the parent class as part of

your own constructor process. Class Object has a default constructor that takes no arguments and which sets the pointer to a reference to Python's Py_None object. So, if you are inheriting directly from Object, this is not a problem. Class SeqBase<T>'s default constructor constructs a reference to a zero-length tuple. In both cases this is a waste of computational effort since the pointer obtained will be immediately replaced by one calculated by the new class' constructor. This is not the only place in CXX where some effort is wasted in order to ensure correctness, and while every effort has been made to be efficient, when in doubt we have chosen correctness rather than efficiency.

3.2 The classes available in CXX

The object hierarchy in CXX is as follows. Inheritance is shown by indentation. Besides the Object family, there is a family of exception classes and some classes to help in the creation of Python modules and extension objects. All names in CXX are contained within the namespace Py.

Object
- Type
- Module
- Integer
- Float
- Long
- Complex
- Char (Strings of length 1)
- SeqBase<T>
  - Sequence (= SeqBase<Object>)
  - String
  - Tuple
  - List
  - Array (NumPy array)
- MapBase<T>
  - Mapping (= MapBase<Object>)
  - Dict

Exception
- IndexError
- RuntimeError
- ... (other errors corresponding to other Python exceptions)

MethodTable
ExtensionModule
PythonType
PythonExtension<T>
ExtensionType<T>

In addition there are a number of functions defined at the global (namespace Py) level. These include the usual binary arithmetic operators and stream output operators.

The documentation shows the tables of methods for each class. Class Object defines a large set of methods that is thereby made available on all of its children. Notable among these are:

- bool accepts (PyObject* p) tests whether or not a member of this class could be constructed using p, that is, whether p points to an object this class was intended to wrap;
- PyObject* operator *() returns the PyObject* contained in this wrapper; this is also available as the result of method ptr().
- Type type() returns the (wrapped) type object associated with this object; a series of queries such as isString () are available to test membership in the standard Python object classes.
- Object getAttr ("name") returns the attribute name of the current object; this is equivalent to Python's ob.name operator.

4.0 Making an extension module

To make a Python extension module is now straight-forward. Let us begin by examining the form to use for a module method.

4.1 Writing a module method

The generic form of the extension module is the same as when using C. First you write a function whose signature is

PyObject* mymethod (PyObject* self, PyObject* args)

In this form, we know that the argument self is unused, and that the argument args is actually always a tuple. We will therefore always have the same structure to our method:

```
PyObject* mymethod (PyObject* self, PyObject* args) {
        Tuple the_arguments (args);
        try {
        ....        do stuff
                    return the_answer;
        }
        except (Exception&) {
                    return Null();
        }
}
```

The try/except clause converts any Python API errors or CXX-detected errors into an exception which is caught in this except clause and converted into a Python exception. (You can also catch the exception instance and clear the exception, as explained in the documentation).

In writing the "do stuff" part of the method, we are now greatly assisted by CXX:

- We can access the i' th argument as the_arguments [i]
- We can affirm the required type of an argument by using it in a copy constructor for the desired type. For example, if the first argument must be a string, we could write:
  String s = the_arguments[0];

This will throw an exception if the first argument is not a string, or if there is no first argument. (Class Tuple also has methods which can check for a certain number or a range of numbers of arguments).

- We don't need to worry about keeping track of any temporary objects in case of errors; cleanup is automatic when the exception is thrown.
- We have direct access to the Python API via the methods of the CXX classes, but in a way that completely ensures correct reference counting. If we use the C API directly we have to make correct use of FromAPI but the need for this is infrequent.
- We can carry out sequence operations in a natural manner, much as if we were working directly in Python, rather than using sequences of Python API calls whose errors must all be checked.

Here for example is a method written using this method that sums the set of float arguments given to it.

```
using namespace Py;

static PyObject *
ex_sum (PyObject* self, PyObject* args)
{
        Tuple a(args);
        try {
                Float f, g;
                int i;
                f = 0.0;
                for (i = 0; i < a.length(); i++) {
                        g = a[i];
                        f = f + g;
                }
                return new_reference_to(f);
        }
        catch (const Exception&) {
                return Null ();
        }
}
```

The function *new_reference_to* (Object *ob*) returns an owned reference to the object *ob*. In this way the Python float object we have created to hold the answer survives the destruction of the variable *f* that occurs when we return from *ex_sum*. If you want a method that doesn't return anything you return Nothing() and to signal a Python exception you return Null().

Note in this example how the assignment g = a[i] not only extracted the i' th argument but ensured that it was a float object. It would have also worked perfectly well not to do this step but directly add f + a[i]. This might be desired, in fact, if you did not want to insist that the arguments be floats.

4.2 Creating the extension module

As usual, we now need an init function for our extension module. In this init routine, which must have C linkage so that Python can find it, we create the extension module and add the method(s) desired to it, as well as any objects we wish to seed into its dictionary (here, as an example, we add the constant *pi*).

```
extern "C" void initexample ();

void initexample ()
{
  static ExtensionModule example ("example");
  example.add("sum", ex_sum, "sum(arglist) = sum of arguments");
  Dict d = example.initialize();
  d ["pi"] = Float(3.14159);
}
```

Note the simplicity compared to writing the same thing in C, with its mysterious "static forward", Python type tables, etc.

5.0 Creating extension object types

CXX also contains a facility for construction of new Python types. This facility is not yet completely satisfactory, but we believe it is a step forward. The key point is that to begin a new Python type we inherit from class PythonExtension, which in turn inherits from PyObject. Thus, at one blow we have made our type a descendant of PyObject, created a type object for it, and created a function that will check whether an object is of this new type. Then we initialize the object in a similar manner to the extension module, adding behaviors and their descriptions. We also write the methods in a similar way.

PythonExtension is a templated class, and the template argument we give it is, shockingly, the class we are defining. This is an example of what Scott Meyers has called the "Curiously Recursive Template Pattern". PythonExtension sets up a Python type object unique to this type, creates a static function

```
static bool check (PyObject *)
```

that tests membership, and sets a deletion behavior that ensures the calling of the class' destructor in the case of its Python reference count going to zero.

Here, for example, is the start of a class "r" defining new objects of type "r" similar to the Python range object:

```
class r: public PythonExtension<r> {
public:
  long start;
  long stop;
  long step;
  r (long start_, long stop_, long step_ = 1L)
  {
```

```
        start = start_;
        stop = stop_;
        step = step_;
    }

    ~r()
    {
        std::cout << "r destroyed " << this << std::endl;
    }
...
```

In a method similar to the way we implemented the module method above, we define the Python behaviors of the new object, such as *r_repr*, *r_getattr*, *r_length*, and methods we choose such as *amethod*, *value*, etc. Then in some module's initialization procedure is a call to *init_rtype*:

```
void init_rtype () {
    r::behaviors().name("r");
    r::behaviors().doc("r objects: start, stop, step");
    r::behaviors().repr(r_repr);
    r::behaviors().getattr(r_getattr);
    r::behaviors().sequence_length(r_length);
    r::behaviors().sequence_item(r_item);
    r::behaviors().sequence_slice(r_slice);
    r::behaviors().sequence_concat(r_concat);
    r::methods().add("amethod", r_amethod);
    r::methods().add("assign", r_assign);
    r::methods().add("value", r_value);
}
```

Extension objects defined in this way have an additional desirable property that ordinary Python extensions do not: they do not have to live on the heap. Of course, as in C++ one must be careful not to retain a reference to a local object after the routine returns. But, with care, one can have the efficiency of objects using stack rather than heap memory. With correct use both new/deleted objects and stack objects will coexist nicely.

At this point it would be nice to somehow get a child of Object, say R, whose job it was to hold pointers to objects of type r. The necessary acceptance test is available as r::check, defined for us when we inherited from PythonExtension<r>. Class ExtensionObject<r> will in fact do this, but it does not contain any methods specific to r. This area and a possible connection to research being done by Geoffrey Furnish at Los Alamos are the subject of future investigations. One goal of this research is to eliminate the need for the try blocks in each method, by hoisting the coordination of Python / C++ exceptions up higher.

6.0 A good match of capabilities

We see that the strengths of C++ are a good match for the weaknesses in the Python language extension process. Use of C++ in this way should lead to much smaller, cleaner, and easier to write extensions, with confidence in their correctness. Reference-counting errors, among the most difficult to prevent and to diagnose when using C, are avoided automatically, even in the difficult case of when an error occurs.

Additional benefits to this approach are the merging of the Python and C++ exception facilities and the ability to write extension modules and objects much more naturally and safely. The facility is completely extensible via inheritance to the users' own classes.

Research remains on further integrating Python and C++ but we believe CXX represents a significant step forward.